Skipove poznamky k predmetu UDDA

skip@realtime.sk

Verzia 0.1

Obsah

Pozn.: Gramaticky urcite nespravne, moze obsahovat chyby, use it on your own risk!

1 GHS

Trosku detailnejsi popis algoritmu GHS.
Predpoklady, oznacenia, pricipy spajania fragmentov mozno najst napr. v slovenskych scriptach k UDDA (nechcelo sa mi to sem kopcit).
Algoritmus:

Na zaciatku:

ak mam level 0, tak najdem najlacnejsiu hranu, oznacim ju "branch of MST" a poslem po nej spravu **CONNECT** + prejdem do stavu *FOUND*.

Ako funguje system hladania najlacnejsej hrany vo fragmente?

Nech ma level L a nech vznikol z 2 fragmentov na leveli L-1. Hranou, ktorou sa spojili nazyvame *core edge*. Prislusne procesory poslu vsetkym procakom v novom fragmente (kazdy do svojej polky) spravu INITIATE. Touto spravou informuje vsetkych o novom leveli a mene fragmentu. Tato sprava obsahuje argument FIND, ktory prepne kazdy procik do dohto stavu.

Ak sa nejaky iny fragment (musi mat level L-1) chce pripojit do fragmentu (proste caka na pripojenie), aj jemu preposleme tuto message, a tak dalej rekurzivne (ak sa do ineho chcel pripojit iny s levelom L-2 …).

Ak procik dostane spravu INITIATE, zacne hladat najlacnejsiu hranu vychadzajucu z fragmentu. Tu je problem, ktora hrana je vychadzajuca a ktora vnutorna (procik to nevie).

Preto si bude processor klasifikovat hrany 3 moznostami:

- BRANCH – hrana, ktora patri sucastnej kostre fragmentu,
- REJECTED – hrana, ktora nepatri kostre fragmentu ale procik zistil, ze spaja 2 vrcholy fragmentu a nakoniec :
- BASIC – hrana, ktora nespada do ani jedneho zo spomenutych pripadov.

Procik hlada hranu vychadzajucu z fragmentu, teda si vyberie najlacnejsiu BASIC hranu a posle po nej spravu TEST(LEVEL, MENO). Ak dojde takato sprava, procik sa pozrie na v akom je fragmente (MENO, LEVEL).

Ak sa mena rovnaju tak posle spravu REJECT a oba prociky si nastavia tuto hranu ako REJECTED. Procik, ktory posielal spravu TEST si vyberie dalsiu hranu a po nej posle spravu TEST.

Ak procik, ktory prijal spravu TEST ma rozne fragment MENO a >=LEVEL tak posle spravu ACCEPT, tj. hrana je vychadzajuca z fragmentu.
Ak ma LEVELrecv<LEVELsend tak *pocka* z odpovedou, dokial jeho level nenarastie. Zdrzanie realizujeme preto, aby ak sa raz fragment recv odlisuje menom, odlisoval sa aj nadalej (v opacnom pripade, ak by ho pohltil vacsi fragment, by sa linky oznacene ako vychadzajuce z fragmentu nejaky cas nedozvedeli o zmene (az dokial nepride sprava INITIATE)).

Takze (kazdy/daky/ziaden) procik fragmentu nasiel vychadzajucu hranu. Teraz o tom musi podat spravu – nespravi to nijak inak ako pomocou spravy REPORT (procesory musia dako kooperovat aby zistili najlacnejsiu hranu).

Ak ziadny procik nema vychadzajucu hranu – hura, vyhrali sme, GG, nasli sme minimalnu kostru (MST).

Inak – vo vseobecnom pripade – kazdy list kostry fragmentu posle po hrane kostry spravu REPORT(W) – W je vaha hrany (alebo nekonecno ak nenasiel ziadnu vychadzajucu hranu). Vnutorne prociky grafu prijimaju reporty, z nich vyhodnotia najlacnejsi a ten poslu po hrane kostry (smerom "do vnutra grafu"). Samozrejme si prislusnu hranu, odkial najlacnejsi report prisiel oznaci a prejde do stavu FOUND (jeho rola v procese hladania najlacnejsej vychadzajucej hrany skoncila). 2 procesory ktore lezia na konci *core edge* (hrana ktora spajala povodne 2 fragmenty) si navzajom poslu REPORT a zistia, v ktorej casti lezi ta najlacnejsia hrana.

V tomto momente mame najlacnejsiu hranu a chceme sa po nej spojit s prislusnym fragmentom. To urobime takto: *core* posle spravu CHANGE-CORE prislusnemu prociku, z ktoreho ide najlacnejsia. To ide, pretoze si v kazdom vrchole kostry pamatame cestu k *nasej* hrane. Ak procik dostane spravu, vysle po hrane spravu **CONNECT**.

SPAJANIE

Ak maju 2 fragmenty levelu L rovnaku najlacnejsiu hranu, obe si navzajom poslu CONNECT. Tymto vytvorime novu *core-edge*, ktore bude niest level L+1. A to sposobi, ze oba krajne procesory novej core-edge poslu INITIATE spravu, kazdy po svojej kostre. Fragment na leveli L obsahuje najviac 2^L, teda $\log_2 L$ je horna hranica levelov fragmentu.

Pozrime sa teraz na pripad ak fragment na nizsom leveli posiela CONNECT fragmentu s vyssim levelom. Ak sa chce fragment F s nizsim levelom spojit s fragmentom FF na vyssom levely a ak FF este neposlal prislusnu REPORT spravu (ze objavil tuto hranu), fragment F sa jednoducho pripoji k FF – fragmentu je cez danu hranu preposlana sprava INITIATE (pozri zaciatok) a pripoji sa k hladaniu najlacnejsej vychadzajucej hrany. Ak procesor z FF na hrane, cez ktoru sa F chce pripojit uz stihol poslat REPORT, tak hrana ma nizsiu vahu ako minimalna vaha hrany vychadzajucej z F a F sa nemusi zucastnit hladania. (report bol o inej hrane akou sa F chce pripojit!)

2 KKM

Vyuziva myslienku uzkeho spojenia dvoch problemov – traverzovania grafu (kostra) a volba sefa. Ak pretraverzujeme graf, procik, ktory inicioval traverzovanie bude sefom.

Problem: co ak je zobudenych viac procesorov a je spustenych viac traverzovani?

Riesenie: algoritmus bude pracovat v leveloch. Procesory si pri prehladavani grafu budu posielat token (iniciator, level).

Myslienka *levelovania* bude nasledovna: ak do procika dorazia 2 tokeny (tj. boli inicializovane 2 traverzovania), tokeny budu zabite (traverzovania abortovane) a vytvori sa nove traverzovanie z tohto procika z tokenom na leveli +1. To sa stane v pripade, ze oba tokeny mali rovnaky level L (podobna myslienka ako v GHS).

Ak dorazi token s levelom L do procika v ktorom je token s levelom L+1, alebo ktory taky token navstivil, token s levelom L je zabity (teda neovplyvni travetrzovanie tokenom L+1).

Token moze byt v 3 stavoch (ANNEXING, CHASING, WAITING) a pouzivame niekolko pomocnych poli (lev[p], cat[p] – iniciator posledneho annexing tokena, wait[p], last[p]).

Algoritmus spolupracuje s traverzovacim algoritmom tak, ze vola funkciu TRAV. Jej vystupna hodnota je ID suseda, kam posleme token, alebo Decide, ak traverzovanie skoncilo.

Na zaciatku je token (Q, L) v mode ANNEXING a poslucha traverzovaci algoritmus – dokial nenastane jedna z nasledujucich situacii:

1. traverzovanie skonci a sefom sa stane procik Q
2. ak dorazi do prociku s vyssim levelom tak je zabity
3. token dorazi do procika kde caka iny token s levelom L. Oba tokeny su zabite a zacne sa nove traverzovanie s levelom L+1
4. token dorazi do procika s levelom L kde bol ANNEXING token s menom (procika) cat[p]>q (tato podmienka urcuje, ze tokeny s iniciatorom, ktory mal mensie ID budu cakat, dohanat ich budu vzdy tokeny s vacsim procesorovym ID – pretoze ich chceme spojit ked uz maju rovnaky level) alebo tam bol CHASING token (token ktory nahana iny token – mozno mna). Token prejde to stavu WAITING.
5. token dorazi do procika s levelom L a bol tam ANNEXING token s menom cat[p] <q. Token sa stane CHASING a posleme ho rovnakym kanalom ako ten predchadzajuci token.

CHASING token nahana token, ktory presiel posledne procikom. Procik ho posiela po kanaly kam poslal ten predchadzajuci. Token si nahana daky token dokial nenastane jedna z tychto situacii:

1. token (Q,L) dorazi do procika p s levelom L<lev[p]. token je zabity
2. token dohoni ☺ proces s WAITING tokenom na levely L. Oba su zabite a zacne sa nove prehladavanie s novym levelom L+1
3. token dorazi do procika na levely L kde bol predtym token CHASING. Token prejde do stavu WAITING.

WAITING proces si kempi v prociku, dokial nenastane jedna z nasledujucich situacii:

- dorazi token s vyssim levelom – nas kemper je zabity
- dorazi token s rovnakym levelom -> nove prehladavanie s levelom +1

Ak traverzovanie skonci, procesor ktory ho inicializoval sa stal sefom a posle broadcast vsetkym ostatnym (som sef, uz nic netreba dalej robit). Hra s levelmi a tokenmi je v podstate o tom, aby sme eliminovali viac naraz spustenych traverzovani. Nakonci prejde do konca traverzovanie s najvyssim levelom a funkcia TRAV vrati *Decide* – si sef! ☺

3 Distribuovana dohoda pri chybach liniek

Problem hladania DOHODY: prociky v sieti zacinaju s nejakou vstupnou hodnotou a vystupom by mala byt rovnaka hodnota (=suhlas. Vstup moze byt ale lubovolny!).

Problem dohody mozeme nazyvat aj Problem Koordinovaneho utoku.
Popisem blizsie:

- Niekolko generalov planuje utok na nepriatela. Utok bude uspesny iba v pripade, ze zautocia vsetci naraz, inak budu ich armady rozprasene. Kazdy general ma na zaciatku svoj nazor na to, ci jeho armada je pripravena zautocit alebo nie.
- Generali mozu komunikovat len prostrednictvom sprav, ktore prenasaju posli. Poslov mozu zajat alebo sa mozu stratit. Generali sa aj napriek tejto nespolahlivej komunikacii musia dohodnut ci zautocit alebo nie.
- Predpokladame, ze graf je neorientovany a spojeny. Tiez pozname hornu hranicu casu, za ktoru bude urcite sprava dorucena.

Ak su linky spolahlive, tak si generali vymenia spravy. Pocet krokov sa bude rovnat diametru (priemeru) grafu. Zautocim vtedy ak vsetci chcu zautocit.

V pripade nespolahlivych liniek je situacia horsia. Neexistuje totiz ziadny deterministicky algoritmus riesiaci dany problem.

Formalny zapis: mame n procikov, poskladane do neorientovaneho grafu, kazdy procik pozna cely graf. Kazdy procik zacina so vstupnou hodnotou {0,1}. Vystup je bud 1 = zautocit alebo 0 = neutocit, abort. Pouzivame synchronny model. Spravy sa pocas putovania v grafe mozu stratit. Cielom je nastavit rozhodovaci stav na 0 alebo 1.

Urcime 3 podmienky pre dohodu:

- SUHLAS. Ziadne 2 procesory sa nerozhodnu rozdielne
- PLATNOST:
    - ak prociky zacnu s 0, tak 0 je jedina vystupna moznost
    - ak prociky zacnu s 1 a vsetky spravy su dorucene, potom 1 je jedina moznost vystupna moznost
- UKONCENIE: vsetky procesory sa rozhodnu

Druhy bod je iba jeden z mnohych, da sa to definovat aj inak (uzitocnejsie pre niektore pripady). Vo vseobcnosti ide o to, aby dohoda (vystup) bola *rozumna.* Zadefinova podmienky su dost slabe – napr. ak jeden procik zacne s 1 tak je povolena dohoda 1.

Veta: nech G je graf z dvoma uzlami, prepojenych jednou linkou. Neexistuje ziaden algoritmus riesiaci problem koordinovaneho utoku (dohody) na grafe G.
Dokaz: obratene.TODO

Tato veta ukazuje zakladnu limitaciu distribuovanych systemov. Avsak v realnom svete musime problem dohody riesit (napr. COMMIT v distribuovanych databazach).

Problem koordinovaneho utoku – Nahodna Verzia

Jednym z rieseni fundamentalneho problemu DS je brat v uvahu pravdepodobnost straty spravy a nechat proces deterministicky. Inym je pouzit znahodnenie pripustajuc istu moznost porusenia podmienok. Pozrime sa na to blizsie:

Kazdy z *n* procesorov zacina so vstupom {0,1}. Predpokladajme, ze proces skonci po konecnom pocte krokov R>=1 alebo inak – v kole R je procik nuteny rozhodnut sa bud 0 || 1.

Ciel je taky isty ako v predchadzajucom, akurat sme si problem trosku oslabili – pripustame (s relativne malou pravdepodobnostou – oznacime ju E) chyby.

Formalne: komunikacny vzor bude podmnozina mnoziny M={(I,j,k) : (I,j) je hrana, k>0} Je to kvazi formalne zapisana komunikacia medzi procikmi (vzor je mnozina trojic!). Dobry komunikacny vzor je taky, kde k<=R. Znamena, ze procik i poslal spravu do procika j v k-tom kole – uspesne.

Definujme protivnika B, ktory sa nam bude snazit situaciu na grafe co najviac stazit. Pre neho definujeme PR^B pravdepodobnostnu funkciu. Preformulujeme teraz problem koordinovaneho utoku:

- SUHLAS: pre kazdeho protivnika B: PR^B <= E
- PLATNOST: to iste co predtym
- Nepotrebujeme podmienku ukoncenia, lebo predpokladame, ze vsetky prociky skoncia do r-kol

Uloha: najst algoritmus s co najmensou hodnotou E a dokazat, ze to lepsie nevyhaxime.

Algoritmus

Zjednodusime si ulohu a predpokladajme kompletny graf o n vrcholoch. Pre tento pripad si popiseme algoritmus, ktory slape pri rovnosti E=1/r.

Definujme reflexivne ciastocne usporiadanie komunikacneho vzoru ako <= kde

1. (i,k)<=(i,K) ak 0<=k<=K (informacny tok jedneho procesu)
2. majme (I,j,k) potom (I,k-1)<=(j,k) (tok z sendera do recievera)
3. (i,k)<=(I,K) a (I,K)<=(II,KK) potom (i,k)<=(II,KK) (tranzitivita)

Pomocou tohto usporiadania si budeme pre kazdy procik drzat level (ten definujeme rekurzivnem, zacina sa na 0). Kazdy procik zacina na levely 0. ak pocuje,ze vsetci su uz na 0, prejde na 1. Atd.
Plati, ze rozdiel v leveloch vsetkych procikov je +-1. V pripade, ze su vsetky spravy dorucene, level ma hodnotu poctu kol.

Idea algoritmu RANDOM_ATTACK je nasledovna:

Kazdy procesor si bude uchovavat svoj level. Procik #1 vygeneruje KLUC – rahodne cele cislo z intervalu [1..r]. tato hodnota je pribalena ku vsetkym spravam. Rovnako vstupna hodnota procika je pribalena k spravam.

Po r kolach sa kazdy proces rozhodne 1, ak vypocitany level je aspon taky velky ako KLUC a vie, ze vsetky prociky mali za vstup 1. Ak nejaka linka zlyhala, level procesora sa nezvysi, lebo nedostal info o leveloch od vsetkych procikov (mame Kn graf).

Veta: RANDOM_ATTACK vyriesi problem znahodnenej verzie koordinovaneho utoku pre E=1/r

Veta hovori, ze pravdpodobnost,ze prociky sa nezhodnu je E. Rovnako vieme, ze rozdiely v hodnotach levelov su +-1. Preto pre vhodne zvolenu kluc, bud vsetky prociky budu bud vyssi level ako kluc, vtedy sa rozhodnu 1, alebo budu mat level mensi, vtedy sa rozhodnu 0. S pravdepodobnostou E zvolime taky KLUC, ze niektore prociky si zvolia 1 a ine 0.

4 Dohoda v distribuovanych systemov so zlyhaniami procesorov

V dalsom budeme rozoberat 2 mozne zlyhania procesorov:
- zastavenie: chybny procik zastane a nic dalej nerobi
- byzantinska chyba: procik sa moze spravat uplne nepredvidavo

Problem dohody: prociky zacinaju so vstupmi z mnoziny V. Vsetky spravne prociky vyprodukuju vystup z rovnakej mnoziny V.

Podmienka platnosti: Ak vsetky prociky zacinaju s hodnotou *v* jedina mozna vystupna hodnota (dohoda) je tiez *v*.

Pocet *chybnych* procikov ohranicime konstatnym cislom *f*.

Siet budeme reprezenovat neorientovanym grafom o *n* vrcholoch, pricom kazdy procik pozna cely graf. Pouzivame *synchronny* model, v ktorom limitovany pocet (*f*) procikov moze *zlyhat*.
Predpokladame, ze linky su 100% spolahlive.

Model so zastavenim procesora
Podmienky pre dohodu:
- SUHLAS: ziadne 2 prociky sa nerozhodnu pre inu hodnotu
- PLATNOST: ak prociky zacinaju so vstupom *v*, potom *v* je jedina moznost vystupu
- UKONCENIE: vsetky korektne prociky sa niekedy rozhodnu

Model s byzantinskymi procesormi
V tomto modeli procesor moze nie len zastavit, ale moze sa zacat spravat nepredvidatelne.
Podmienky pre dohodu:
6. ziadne 2 *korektne* prociky sa nerozhodnu pre inu hodnotu
7. ak vsetky *korektne* prociky zacinaju so vstupom *v*, potom *v* je jedina moznost vystupu pre *korektne prociky*
8. vsetky korektne prociky sa niekedy rozhodnu

Vstah medzi Byzantinskym problemom a problemom zastavenia: Algoritmus, ktory riesi Byzantinsky problem automaticky *neriesi* aj problem zastavenia. Vypliva to z definicie podmienky SUHLASU. V pripade zastavenia sa musia rozhodnut pre rovnaku hodnotu vsetky prociky, aj tie, ktore medzicasom zastavia. Ak by sme nahradili podmienku SUHLASU v probleme zastavenia Byzantinskou, implikacia by platila. Alebo, ak sa vsetky korektne prociky v Byzantinskom algoritme vzdy rozhodnu naraz, tak je to tiez OK.

Algoritmy pre STOP-CHYBY

Predpokladame komplenty n-vrcholovy graf.

BASIC ALG – FLOODSET

Jednoduchy algoritmus, ktory broadcastuje vsetky hodnoty, ktore kedy videl. Kazdy procik spravuje pole W (podmnozina V, V=vstupne hodnoty). Na zaciatku W obsahuje iba hodnotu vstupu W. V kazdom z f+1 kol, kazdy procik spravi broadcast W. A do W si prida vsetky hodnoty, ktore prijal. Po f+1 kolach aplikujeme nasledujuce pravidlo: Ak W obsahuje jeden prvok, tak ten je vystupom. Inak sa procik rozhodne pre v0 (v0 = defaultna vstupna hodnota v mnozine V).

ZLOZITOST algoritmu

Cas: f+1 kol
Spravy: O((f+1)n^2)

EIG Algoritmy (Exponential Information Gathering)

V tomto type algoritmov si procesory posielaju spravy, ktore si potom ukladaju v datovej strukture zvanej *EIG strom.* Na konci sa rozhodnu podla rozhodovacieho pravidla, ktore aplikujeme na hodnoty zaznamenane v strome.

EIG strom pozostava z urovni 0 … (F+1); Root ma prazdny string “lambda” a kazdy otec na *k-tej* urovni ma *n-k* synov.

EIG algoritmy su vo vseobecnosti narocne co do poctu poslanych bitov (sprav) po sieti, aj co do poctu lokalneho ulozneho priestoru. Touto datovou strukturou vsak vieme riesit Byzantinsku dohodu.

EIG Stop ALGORIMUS

Kazdy procik bude mat vlastny EIG strom. Kazdy vrchol si bude drzat 2 udaje: meno (x), hodnotu (v=val(x)). Algoritmus pracuje v F+1 krokoch (0-ty krok je inicializacia vstupov do stromu). Procesory si budu posielat informacie – co vedia o vstupoch vsetkych procesorov (predpokladajme, ze procesor moze poslat spravu aj sam sebe kvoli jednoduchosti).

Na zaciatku kazdy procesor posle vsetkym svoj vstup. V dalsich k-kolach bude procesor i posielat dvojice (x,v), pricom x je menovka na k-1 urovni neobsahujuca i. Procesor j prijme od i dvojicu (i1..ik,w) a bude vediet, ze ik mu povedal, ze i(k-1) povedal ik, ze … i1 povedal i2, ze jeho vstup je w. Takto sa procesory metodou “ukecanych babiek na trhu” dozvedia o vstupoch procesorov (aj chybnych, ktore svoj vstup stihli povedat iba niektorym procikom – nevadi, nevadiiii, ved ostatni sa to od nich dozvedia). Do vrchola, kde maju byt udaje od chybneho procesora za bude zapisovat NULL (ak mi ten procik nic nepovedal).

Na konci F+1 kroku aplikujeme rozhodnutie – oznacme W mnozninu vsetkych hodnot v strome. Ak W je jednoprvkova mnozina (vsetci mali rovnaky vstup) tak ako vystup zvolime tuto hodnotu v. Inak zvolime v0 (nejaka default z mnoziny vstupnych hodnot V). Teda ak zlyhal daky procik a dostali sme niekde null, tak sa o tom vsetci isto dozvedia (v tom F+1 kole najhorsie). Chybny procik mohol mat na vstupe nejaku inu hodnotu, preto

sa vsetci musia rozhodnut pre default v0 hodnotu aby sme zachovali 2. podmienku dohody – platnost.

ZLOZITOST algoritmu
Cas: f+1 kol
Spravy: O((f+1)n^2)

Algoritmy pre Byzantinske chyby

Rovnako uvazujeme Kn graf a navyse predpokladame, ze n>3f (pocet procesorov je 3x vacsi ako chybujucich). *Byzantinska dohoda* je komplikovanejsia ako dohoda pri *zastaveni*. Konkretne: 3 procesy nedokazu vyriesit problem byz. dohody, ak je i co 1 z nich byzantinsky (dokaz vid. obrazky v slovenskych scriptach).

EIG Algoritmus pre Byzantinsku dohodu (EIG –Byz)
Predpoklad n>3f. Algoritmus pouziva rovnaku datovu strukturu – EIG strom ako algoritmus EIG-Stop.
Strategia propagovania informacii je rovnaka – az na to, ze chybnu spravu si bude procesor opravovat tak, aby vyzerala zmysluplne. Rozhodovacie pravidlo bude ine, pretoze niektore procesory si mozu vymyslat a tak korektne procesory tieto chyby musia dako zamaskovat.

Procesory posielaju spravy rovnako ako EIG-Stop, F+1 kol az na nasledujuce vynimky:

4. ak dostane spravu ktora je uplne nezmyselna (garbage, duplicitne hodnoty) tak tuto spravu jednoducho zahodi. Tvari sa ako keby od toho procika nic nedostal.
5. na konci kola f+1 kazdy procesor nahradi vsetky hodnoty NULL vo vrcholoch EIG stromu hodnotou v0 (default vstup).

Aby sa procesor spravne rozhodol, priradi kazdemu vrcholu este jednu hodnotu. Bude postupovat od listov ku korenu:

6. pre kazdy list s menom x priradi newval(x) = val(x)
7. vsetkym ostatnym vrcholom s menom x priradi newval(x) definovanu nasledovne: newval bude jednoznacna vacsina hodnot deti vrchola, ktora je z V (mnozina vstupov)
8. ak neexistuje vacsina tak priradi v0

Procesor *i* sa rozhodne nakoniec pre newval(root)

Narocnost: CAS: F+1 **SPRAVY:** O((F+1)n^2)

TurpinCoan Algoritmus pomocou Binarnej Byzantinskej Dohody
Pomocou tohto algoritmu pre mozny vstup obmedzeny na {0,1} sa da riesit vseobecny problem Byzantinskej dohody (2 kola navyse, 2n^2 sprav navyse). Tymto dokazeme usetrit vela bitov v prenose po sieti, pretoze nemusime prenasat cele hodnoty, ale staci nam jeden bit.

ALGORITMUS:
Kazdy procik ma 4 premenne: *x,y,z* a *hlasuj*. *x* je inicializovana hodnotou vstupu.

Kolo 1: kazdy procesor I posle svoj vstup (premenna x) vsetkym ostatnym (aj sebe). Po prijati vsetkych sprav: ak prislo >=n-f nejakej hodnoty z V (vstupu) tak si *y* nastavi na v, inak na null.
Kolo 2: kazdy procesor posle vsetkym (aj sebe) hodnotu *y*. Ak dostane >=n-f sprav nejakej hodnoty, tak nastavi *hlasuj* na 1, inak na 0. Zaroven do *z* nastavi spominanu vacsinovu hodnotu (ak su vsetky spravy null tak z zostava nedefinovane).
Kolo r: procesor spusti *proceduru* binarnej byzantinskej dohody, ako vstup pouziva obsah premennej *hlasuj*. Ak sa procesor rozhodne pre 1 a premenna *z* je definovana, potom finalne rozhodnutie je *z*. Inak v0.

Zlozitost:
Cas: r+2, kde r je pocet kol binarnej procedury
Spravy: to co binarna procedura +2n^2 sprav.